Flight Control System for an Autonomous Unmanned Aerial Vehicle

Daven Hiskey and **Jianna Zhang**

Computer Science Department
Western Washington University
516 High Street, Bellingham, WA 98225-9062
daven@hiskey.us, jianna.zhang@wwu.edu

Abstract

This paper deals with developing a flight control system for a small Autonomous Unmanned Aerial Vehicle. A flight control system for an Autonomous Unmanned Aerial Vehicles has numerous applications in real world situations; such as search and rescue missions, reconnaissance, and other such military and domestic applications that have real potential to save human lives. This paper on developing a flight control system includes covering computing hardware, sensors, plane development, flight control design and implementation, and testing the system as a whole in a custom simulation environment.

Keywords: Autonomous Unmanned Aerial Vehicle (AUAV), Aircraft Control, Flight Control System

1. Introduction

This paper involves investigating and developing a robust flight control system that is capable of being used on a real world Autonomous Unmanned Aerial Vehicle (AUAV). In addition to this, we cover hardware and plane requirements including setting a goal of keeping the chosen hardware/plane relatively inexpensive in order that others can more easily build off this project without needing significant funding.

AUAVs need the ability to make intelligent decisions in a dynamic real world environment with real world sensor noise and potential failure. These systems must be able to adapt to these situations and still perform their primary objective with a high rate of success. This paper covers developing such a system that could be implemented easily on a real world AUAV. Covered in this project are what kind of plane should be used for ease of development and robustness in terms of steady flight, durability, and weight carrying ability. We also look at hardware needed for this application factoring in the weight / speed tradeoff of the computing unit(s) needed to run the flight control software. Next, we will look at the actual design of the flight control system itself; designing it in such a way as to be easily extensible for a variety of objectives or "missions". Finally, we look at developing a simulation environment for testing the system, which is necessary in this case as testing while flying is not a good way to do things in terms of expense (crashes) and actually being able to tell easily where problems may develop in the flight control system's implementation

.

2. Background and Approach

Unmanned Aerial Vehicles (UAVs) have been around for a long time in some state or another. What we typically think of as UAVs in modern times is remote control aircraft, both large and small. With recent technological advancements, people have begun to attach various sensors to their UAVs, such as cameras, GPS, and other sensory hardware. This has allowed people to fly their UAVs outside of their field of view. These types of set ups are being primarily used for military operations and by advanced RC plane hobbyists [14] [15].

Further advancements have begun emerging recently where autopilot software has been installed on UAVs to allow the UAV to hold steady flight and to travel from waypoint to waypoint with little to no human interaction; not too dissimilar to what is done on commercial airliners, but with the noted difference in that many AUAVs are very small and have limited processing power, weight carrying ability, and electrical power at hand over the larger airliner counterpart systems. As such, developing a robust flight control system for small AUAVs presents unique challenges over their larger counterparts.

The current state of this type of solution on small scale AUAVs still requires a human or several humans to closely monitor the AUAV and at times take control of the aircraft. These AUAVs primarily use hard coded solutions to fly themselves. As a result of this, these robots generally do not typically have the ability to fly in complex environments, such as heavily wooded or urban type environments, without having a human monitoring the plane in some way and possibly taking control at times [1].

As such, much research is currently being conducted using machine learning algorithms to AUAVs [1] [2] [3] [9]. These machine learning approaches are much younger than the current hard coded solutions and have very few "real world" implementations; rather, most of the machine-learning solutions have been tested only in simulated environments with artificial noise. This is primarily due to the afore mentioned fact that most AUAVs are small with very limited processing and electric power due to their size and lift potential. However, these solutions do show a great deal of promise in simulation [1], simply requiring some additional advancements in electrical efficiency and size of current high end processors in order to be more viable on real small AUAVs.

In this research project however, testing using machine learning algorithms on the best hardware available that fit our limited weight/size/power requirements showed that real time machine learning during flight was not yet feasible at any great extent in terms of the AUAV being able to learn from mistakes and react quickly enough to recover given the speed and altitude of the aircraft. This, combined with the fact that one of the goals of the system developed here was to be able to eventually run this system on a real world plane, caused us to abandon the machine learning approach in favor of a "hard coded" flight control system solution. However, we still designed it in such a way that it would be relatively easy to add learning elements to the flight control system as hardware advancements make it more feasible for real time, in-flight, learning.

3. Vehicle Platform

In developing a flight control system that can be used on a "real world" AUAV, we needed to first pick out a plane that could be used so that we could determine our weight, size, and power requirements. In addition to this, so that we could accurately factor in the speed in which things would have to happen given the plane's stall speed and factors such as these to make sure that the system would be able to react fast enough to maintain safe, steady flight, given the weight, processing power, and speed of the aircraft.

The important elements that were factored in on selecting an aircraft type were first, that we wanted it to be as cheap as possible to allow others to build off this project without needing extravagant sums of money as in many other projects of this type [8] [12] [13]; We also wanted as much lift as possible to accommodate the added hardware and added power supplies to the plane. In addition to this, it had to be big enough to have a place to put this hardware while maintaining the aircraft's center of gravity for optimal maneuverability and steady flight.

After extensive research and consulting with many experts in the field of remote control aircraft, we determined the "flying wing" style craft would be best for our needs. It gives us ample space to place the hardware we needed and maximal lift for the size. In addition, if designed and built well, it can also absorb crashing without significant damage better than many other styles of RC aircraft.

Given our somewhat stringent requirements, we chose to build this aircraft ourselves having only the EPP foam wing cores themselves pre-cut for us. The wing core design we used was a RitewingRC TL-60 lightning cut (lightning cut specifying more lift than the normal TL wing core design; the "60" signifies the wing span is 60 inches from tip to tip).

We then constructed the plane from scratch using the pre-cut EPP foam wing cores and a variety of other materials, following guidelines and tips received from various RitewingRC and flying wing style plane builders.

With this type of design, we were able to hollow out a large percentage of the center of the plane with the hollowed out portion extending out into the wings several inches on each side; all the while still maintaining the planes optimal center of gravity and structural integrity. One of the benefits of this style of plane is indeed that it maintains its structural integrity very well despite having most of its center hollowed out. This is accomplished by adding carbon spars, fiber glass fabric, and thick layers of 3M 90 glue to attach the fiber glass fabric.

The end result of all of this was that we were able to get enough volume hollowed out in the plane to allow for a small general computer, some circuitry, two power supplies, and some other miscellaneous electronics as laid out in the next section. For full details on the plane design, building, and to see pictures see our notes at the end of this paper.

4. Hardware

One of our original goals in this research project was to allow this system to be very extensible including possible real time image processing requirements and things of this nature. As such, we sought to find the best computing hardware available that fit our weight and size requirements. Settling on just a PIC or other microcontroller type alone wasn't a good option for this type of setup given the high speeds decisions need to be made in and factoring in potential future upgrades we would like to make as stated previously.
So for the main computing piece of hardware we settled on the PICO-ITX mini "general" computer. This unit has a 1GHz single core processor, 1GB DDR RAM, and a

relatively tiny form factor given the computing power measuring at 10 cm x 7.2 cm x 2.54 cm (with heatsink attached, without at 1.2 cm thick). This is roughly the size of a playing card in width/length. This computer also notably features USB and serial connectors which are necessary in this project. The power consumption of the PICO-ITX is roughly 9-20 watts depending on usage, with typical usage around 14 watts. So the size, weight, and power requirements make this "mini-computer" a very nice fit for this type of application.

The hard drive unit attached is a small 2.5 inch SATA hard drive, though in future work, we are planning on switching to using a small USB flash drive for increased power efficiency and weight/size attributes.

We also use a Basic Micro Atom Pro 24 microcontroller for handling all control of servo motors, main motor, and dealing with sensors. Communication between the microcontroller and the main computer is handled via a serial data line and a custom communication object and protocol made for this application as outlined in section 5.3.

The servo's themselves are fully digital HiTech ultra torque HS-5645MG used for controlling the elevons, which are the piece of the plane which controls pitch and roll. These digital servos can be controlled very accurately through pulse-width modulation. A high torque servo is needed in this case due to the weight of the aircraft as well as the speed capabilities of the plane (as much as 100 mph).

The main motor controller used is a Castle Creations Phoenix 80 brushless motor controller. This piece of electronics itself has a small microcontroller built in along with some other hardware. It accepts pulse-width modulation at various widths to control the speed of the motor and also regulates power to the custom circuit added for this project and to the servos. It also handles regulating the required amperage / voltage to the motor. It achieves changes in speed of the motor by switching on and off the motor around 13,000 times per second give or take depending on requested speed.

The specific motor used is a ARC 28-47-1 with a rated speed of 35,000 RPM's (51,000 peak) and a recommended output of 500 watts (50 amps). We also added a 3.33:1 gear box to get a little more power out of it for the extra weight we carry on this plane over a "standard" RC setup.

This flight control system does not handle taking off and landing. As such, and for safety reasons, we include the other typical hardware found in RC aircraft; namely a receiver and transmitter unit. In this case, we used a SUPREME IIS 8 channel receiver in tandem with a 7 channel transmitter for manual control during takeoff and landing as well as possible manual control if there is a software failure during flight. This transmitter/receiver pairing has an effective range of 1-2 miles.

The only sensors needed by this flight control system are a GPS sensor (in this case we are using a parallax GPS unit, but type in this case is somewhat arbitrary just so long as it's small, light weight, and is capable of tracking a relatively large number of satellites to maintain a good signal). We then used a 2D accelerometer for gauging pitch and yaw of the aircraft. In this case, we used an ADXL202EB accelerometer module which was easily connected to our circuit and allows for simple serial communication with the Basic Atom Pro 24 microcontroller.

Initially we thought to add other sensors to be able to handle possible failure of the GPS unit. However, given one of the goals of the project was to keep the price as cheap as possible, it was determined that no more sensors are strictly required for the scope of this project. Failure of the GPS unit isn't likely for any extended period given that the plane will be flying above obstructions and that the particular GPS unit chosen is capable of tracking numerous satellites at once. In order to compensate for potential brief loss of the GPS signal, the flight control software will cause the plane to fly in a circle at a steady altitude until the GPS signal can be acquired again. This assures the plane will not be diverted too far off it's course or potentially crash by lack of GPS altitude data.

In future work, we would like to add a few small camera units for allowing the AUAV to perform more diverse missions than currently possible with the basic flight control system and sensor package. Planning to add sensors like this also influenced the decision to develop the system using a more powerful "main computer" over just a microcontroller, as microcontrollers are poorly suited to process visual information real time.

The power supplies used here are two 11.1 volt lithium polymer 3 cell, 5 amp, battery packs. One of the packs is used to power the motor, servos, sensors, receiver, and microcontroller. The other pack is used to power the main computer. Flight time for this setup can range between 15-25 minutes off these power packs.

Again, for full details on the hardware and plane setup including building and putting everything together, see the notes at the end of this research paper. The above is more meant as a general picture of the ideal setup involved for this system; though the flight control system is designed in such a way as to not specifically need to be tied to one plane/hardware setup. By design, with very minor modification it could be adapted for other aircraft types and hardware.

5. Flight Control System

5.1 Introduction

In developing this flight control system, a principal goal was to design the system in such a way as to make it as extensible as possible. Not only being able to relatively easily adapt it to various "missions" or objectives of the flight, but also to be able to add elements to the system without needing to re-write other core elements implemented here. This system also needed to be fast enough to accommodate the very quick decisions the AUAV would need to make. The plane used in this case can travel as fast as 100 mph, for instance, and the flight control system needs to be able to react fast enough to maintain safe flight patterns and achieve mission goals given this extreme speed. We were however willing to sacrifice efficiency somewhat, where necessary, in favor of ease of extensibility.

To meet the extensibility requirement, we chose to implement the flight control system in a hierarchical, layered fashion [5] [8] with each layer having a clearly defined method of interacting with the other layers in the system. In this way, layers can be added at arbitrary levels to perform different tasks, as well as changing mission objectives without need of modifying other existing layer elements.

The main layers included in this basic flight control system are as follows: hardware and hardware control layer; communication layer; general navigation layer; and a mission control layer. We also added a simulation layer on top of all the other layers, though it's not strictly part of the flight control system, but is rather used for ground based testing as outlined later in this paper.

Some future layers we look to add are an obstacle avoidance layer as well as an object recognition layer for search and rescue and reconnaissance style missions. Given the design of the system, both of these layers can be easily added to the system without needing to modify other layers.

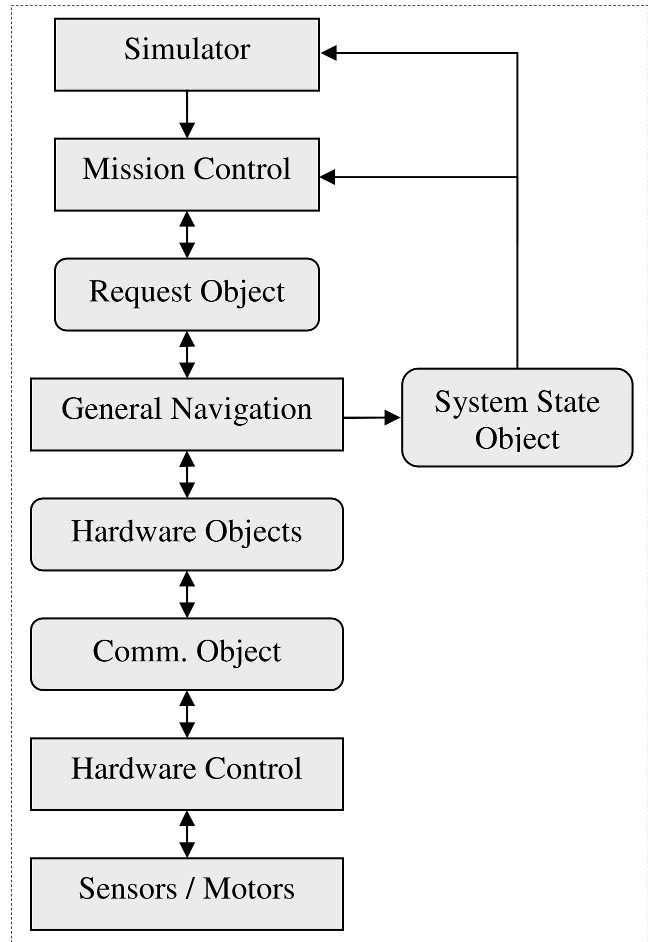

Figure 1: General outline of the main elements of the flight control system and their basic interactions with each other.

5.2 Hardware Control Layer

The hardware and hardware control layer itself is comprised of the hardware outlined in section 4 of this paper, including the microcontroller. This layer is in charge of actually dealing with the sensors in terms of retrieving the data and translating it into a form expected by the flight control system. This layer is also in charge of controlling the main motor, servos, and translating requests from the flight control system software into necessary outputs expected by the motor controller and servo motors.

By having this microcontroller middleware, we make it easy to change the hardware itself without having to modify anything but this layer. As far as the rest of the flight control system is concerned, the requests sent are the same regardless of what the pulse width is expected by the servos for a certain setting of the servo. This allows for easily adapting this system to a completely different plane type and hardware than built with this project.

One thing to note about servo motors and the main motor controller is that both expect to be continually "pulsed" with specific pulse widths in order to maintain their state. These updates must come continually every 20ms or less. One problem encountered is that the single threaded microcontroller must be continually listening for requests from the above layer, but at the same time must be updating the servo's and motor controller in under 20ms between updates. For the servo's, this isn't as much of an issue if the updates come a bit late occasionally because it will then just adjust to what is requested and the flight

control system will recover from any pitch roll changes encountered due to pulsing out of the specified time period. For the motor controller however, this is a very big issue if the updates come outside of this range. This is because the motor controller has a built in cut off if it stops receiving updates. This is a safety feature for if the plane goes out of range of a transmitter or for some other reason loses a signal. Once this cutoff happens, the motor controller must be re-initialized by pulsing at a certain rate outside of the normal pulse range for controlling the motor.

What this then all means is that we somehow had to find a way to have the single threaded microcontroller continually waiting for requests from the main flight control system, while also making sure to pulse the servos and main motor every 20 ms at most. We achieved this by including in the communication protocol between the main computer and the microcontroller middleware a "null" request, which does nothing, but causes the microcontroller to run through its request handling code and during this step pulses the servos and motor with the last requested settings for those servos and motor. We then have the main flight control communication layer continually sending null requests if there are no other requests from the above layers.

Another method to get around this problem we tried was to see if the microcontroller itself could just include a 20ms timeout on waiting for requests and then go on to processing the pulse functions and finally go back to listening. In practice however, we found that this resulted in very long times for the microcontroller to receive/acknowledge requests; sometimes as much as 20 to 30 seconds before the microcontroller would finally be in the appropriate state to "hear" the requests coming in. This is because the microcontroller itself contains no facility to buffer serial data requests; if the microcontroller isn't listening on the correct port at the time serial data is sent, it won't ever receive it. The microcontroller does have a serial buffer on the hardware port used for initializing programming of the microcontroller, but it was recommended that we not use this, as it leaves potential for resetting and otherwise messing up the program loaded on the microcontroller.

5.3 Communication Layer

In order to interact with this hardware layer, we have made a communication layer and a basic protocol for making and fulfilling requests by above layers. In addition to this protocol, the communication layer is comprised of a special object which handles connecting serially to the hardware layer and continually making requests of the hardware layer to assure the servo's and motor are being continually pulsed. If there are no requests from above layers, it simply makes a null request of the hardware layer.

The communication protocol used here in the communication layer between the microcontroller and the above layers is primarily comprised of control codes shared by both sides and a "call and response" type interaction to verify both sides are talking correctly. The exact method of interaction is as follows. The main flight control system will send a control code (single ASCII character) along with possible data if the request is for setting speed or the elevons at a certain level. The microcontroller software then takes this request code and processes the request accordingly; then either sending an "ACK" ASCII code back saying it received the request and processed it successfully or if data was requested from the sensors it will send the same received control code back, thus identifying the data type, and then will send the translated data from the sensors.

For instance, a typical request for the elevons goes as follows. The main flight control system will send the appropriate ASCII character associated with the left elevon, followed by the requested degree to adjust the servo (0-180 degrees, with 90 degrees being centered). The microcontroller will receive the control code followed by the byte of data (0-180). It will then translate this degree into the corresponding pulse width value needed to pulse the servo with to achieve the expected result given the specific servo motor pulse-width range values. It then sets this pulse width for future use, so the plane will maintain this degree level until a new left elevon setting is requested by the main flight control system. The microcontroller then handles "driving" the servos and motors by pulsing them with the stored settings from requests. It then in this case sends a special "ACK" ASCII character back to the main system to acknowledge that it received the request and fulfilled it. Finally, with the request completed and response sent, it sits and waits for another request.

In the case of a sensor request, the main flight control system will send a request for some specific sensor data, for instance, the current pitch of the plane. Using the associated ASCII control character code, the microcontroller will then take that control code and then do the appropriate thing to talk to the 2D accelerometer and then translates the returned raw data from the accelerometer to the form the flight control system expects, in this case -90 to 90 degrees. The actual accelerometer data looks very different from this output form, but is calibrated appropriately in the microcontroller middleware software. Once again, the microcontroller at this point will pulse the servos and motor before sending back the response to the main system. Then, once the data is translated and the servos/motor pulsed, the data is then sent back in the form of: "ASCII control code" "data"; in this case 2 bytes are sent back, repeating back the same ASCII control code sent signifying pitch and then the -90 to 90 value in byte form.

The full list of requests available in this system setup is as follows; from the GPS: number of satellites currently tracked; latitude; longitude; altitude; speed; and heading; from the 2D accelerometer, X and Y (pitch / yaw) values. Next, from the microcontroller settings itself, motor speed (scaled to 0-10) and left and right elevon settings (translated from pulse-widths to 0-180 degrees, 90 degrees being centered). Finally, from the receiver, whether the plane is “computer controlled” or not. Or in other words, whether it should be flying itself or whether it is being manually controlled as in a “normal” UAV.

In order to determine this control, there is a free channel on the receiver/transmitter that is used to determine and change who is in control of the plane. In this case, depending on whether a switch on the transmitter is flipped on or off. From this, there are a couple ways of handling how the system deals with changes in who is in control. In our case, we chose to use relays in our circuit and have the microcontroller switch control based on this. The microcontroller can still send out requests, but because the relay is flipped the requests won’t ever get to the servos / motor; rather the receiver will be picking up and fulfilling the requests from the ground based controller through the RC transmitter. Similarly for the computer controlled case, if the switch is flipped such that it should be “computer” controlled, then the microcontroller flips the relays such that the requests from the microcontroller will be fulfilled again and the requests from the ground based controller ignored, excepting the control switch request which is detected by a port on the microcontroller through a channel in the receiver.

The alternative would be to have the middleware layer receive the requests from the ground based controller and relay those requests instead of the flight control system’s requests; but with this method, you’d then need to translate appropriately the requests coming in from the receiver into output commands that the microcontroller is capable of producing. This is entirely possible and not prohibitively difficult, but seemed like more work than the other method and needlessly making the microcontroller a middleware of a system that really needed none (the standard RC transmitter / receiver combo).

5.4 General Navigation Layer

The next layer above the hardware and communication layer is the General Navigation Layer (GNL). This layer is in charge of fulfilling request from above layers, while still keeping the plane in the air flying safely. It is in charge of things like recovering from stalls, recovering from flips, maintaining a safe altitude, and generally keeping the plane out of dangerous flight states. Doing all of this while it is also translating requests from above layers into actual actions that the plane is capable of taking while flying safely [8] [12]. Requests from above layers come in the form of a request object. The information included in the request object is as follows: speed, altitude, latitude, and longitude. It also includes a “completed” flag which is the mechanism in which the above layers can determine when the request has been accomplished and can thus move on to the next request.

The general flow of execution here with the GNL is as follows: if there is no current request to the GNL from above layers, it just controls the plane such that the plane continues on as it has been. In this state the GNL has only two jobs. First is to make sure the plane keeps flying safely; handling things like making sure the plane isn’t stalling, isn’t flipped over from a gust of wind or for whatever reason, and generally keeping flight as steady as possible by keeping the plane within acceptable pitch/yaw limits and the like. Second, the GNL makes sure the plane isn’t flying outside of pre-set safe boundaries. If the GNL detects that the plane has gone outside of this range, it will re-direct the plane back into those pre-set boundaries. This mechanism assures that if there are no requests for a long period that the plane won’t just fly off in some direction until running out of power.

Concerning handling of requests, when the GNL receives a request, it will then figure out what adjustments it needs to make to the plane’s flight to fulfill the request. For instance, if the request includes a speed value different than the current speed, it will make the appropriate request through the communication layer to the hardware layer to adjust the speed as requested. It will in a similar fashion make altitude adjustments as needed. Finally, if the request includes a lat/lon coordinate it will figure out what heading it needs to turn the plane to so that it will eventually reach that lat/lon coordinate. Through all of this, it will still continually check to verify the plane is still flying safely and still heading towards the requested coordinate if any. Once it reaches the coordinate and has made speed or altitude adjustments if any, it marks the request as completed and continues flying as illustrated previously until a new request is detected and then it will go about processing that request.

In order to actually get the plane to fulfill the requests and do the needed action, the GNL contains hardware objects representing the types of requests that can be made to the actual hardware using the communication layer. These two layers combined allow the GNL to interact with the plane’s hardware in an abstract way. Each hardware object represents a real piece of hardware (sensors / motors / servos). These objects contain methods outlining all available interactions with the real hardware. For example, with the elevon object, the available methods are retrieving the centered position (90 degrees); max position (180 degrees); min position (0 degrees); getting the current right and left elevon positions; and setting the right and left elevon positions; the latter get/set methods being used in

tandem with the communication layer object for actually fulfilling the request.

For a specific example of how this interaction takes place, we will now look at how the system handles making an altitude adjustment. The GNL would first request from the GPS object the current altitude of the plane using the "GetAltitude" function, passing the GPS object method the serially connected communication object. If the received altitude value is not within a set threshold with respect to the requested altitude, it will then determine it needs to make an adjustment.

To accomplish this adjustment, it will slowly adjust the elevons using the elevon object's "SetElevons" function and again passing it the connected communication object along with the degree values to set the right and left elevons at; adjusting the specific values for setting the elevons (0-180 degrees, 90 being flying level) according to whether the plane needs to increase or decrease altitude.

It's then up to the hardware layer to determine how to actually get the elevons to do the requested action by manipulating the servos accordingly through translating the degree value to a useable pulse width value which the servo motors respond to. The GNL will then continue to adjust the elevons as needed to achieve the requested altitude, while still monitoring that the plane is always within acceptable ranges of pitch/yaw and keeping stable flight. In particular, in this case the max pitch value is essential to ensure that the plane doesn't just do a loop in the air, but rather will reach that max pitch value and adjust the elevons such that it never goes above/below this; thus will continue to increase/decrease altitude rather than doing a loop.

Eventually the requested altitude will be reached, within some set threshold of a few feet. Once this happens, assuming there are no speed or lat/lon coordinate requests associated with the request that have not been completed, the request will be marked as complete and the mission control can then move on to the next request if programmed to do so.

It should be noted here, that at any time during a request the mission control or any other object passing the request to the GNL can change the request if it chooses to do so before the request is complete. This will not adversely effect the GNL as it will then just adjust what it is doing based on the new request.

5.5 Mission Control Layer

Above the general navigation layer can be an arbitrary number of layers performing various tasks. In this basic case, we simply have a mission control layer. This layer simply creates requests based on requirements of the mission and passes them to the GNL. This layer does not have direct access to the hardware layer. It can however monitor the state of the plane through a special hardware data state object which is updated continually as the GNL makes requests. This object contains a cache of the most recent known sensor data, control state, motor state, and elevon states. It then can react to this data by making/modifying requests if necessary or can ignore the system state and simply cycle through its pre set requests such as in the case of way point navigation type missions [5].

Missions themselves are extremely easy to design and implement, simply needing a set of request objects comprising speed, altitude, latitude, and longitude values. As indicated, more advanced functionality can be implemented, with monitoring the state of the plane and modifying requests if needed for the specific mission, but in it's most basic state the mission object simply needs a list of request objects and a method to cycle through them waiting in turn for each to be complete before sending the next request. The GNL handles all other aspects of maintaining safe flight without needing the mission control object to do anything but make requests for the location it wants the plan to fly to and at what altitude and speed to maintain. In the latter two cases, the GNL may override requested altitude and speed settings if it is deemed unsafe given the plane's state; such as if the request specifies an altitude below the set safety threshold or if the plane appears to be stalling and the GNL increases speed among other things to return the plane to a safe state.

With this layered design and the set standard of using a request object to pass requests between layers and using the cached hardware state for layers above the GNL to be able to access the system state, it is very easy to add additional layers and plane functionality between the mission control layer and the general navigation layer. For instance, one future addition that we would like to make on this system is to add an obstacle avoidance layer. This layer would then take the request from the mission control layer, check the state of the plane in terms of whether or not there is an object in front of the plane that it is imminently going to hit. If so, it can store the request from the mission control and rather forward a different request to the GNL that will cause the plane to avoid hitting the object in front. Once this request has been completed, it can then pass on the previous request from the mission control layer or a new request from the mission control layer if the mission control has changed the request based on the new state of the plane.

In this fashion, an arbitrary number of layers can be inserted between the mission control layer and the GNL using this uniform request object. The higher priority something should have, the closer to the GNL it should be placed as requests get passed down through the layers. The limit to the number of layers then is only based on

speed of the system and number of calculations required by each layer on down in terms of making adjustments quickly when necessary. This very modular approach also lends to being able to replace existing layers easily without needing to modify other aspects of the system. Such as possibly at some point being able to replace the GNL with a perhaps more robust continually learning version of the same software once the needed computational power is available given the size/weight requirements of this application.

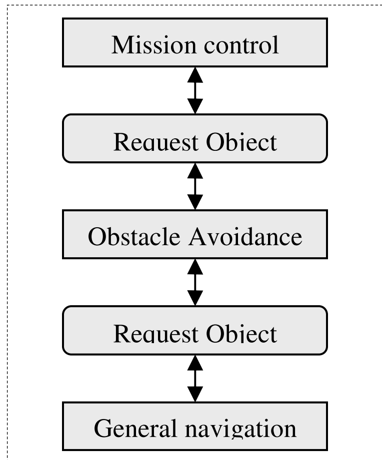

Figure 2: Example of flight control system with a new layer added (Obstacle Avoidance).

The final layer in this system is the simulation layer used for testing; which strictly speaking is not a part of the flight control system and will be covered in the following *Testing and Results* section.

6. Testing and Results

6.1 Introduction

Originally as stated previously, we had planned on having the general navigation layer be coded in such a way as it would be continually learning how best to fly the plane and keep it stable while still fulfilling requests. The training examples used then to teach the plane how to fly initially would be accomplished by flying the plane manually and having the GNL learn from example how to fly safely, in a similar fashion as has been done in numerous other learning robots such as Stanford's DARPA challenge champion car "Stanley".[14] However, computational power became an issue here in terms of being able to do this real time.

As such, we abandoned the learning methods for the GNL and instead needed a good way to adequately test our flight control system without actually needing the plane to be flying. For obvious reasons, testing in flight was not an optimal solution. Instead, we settled on creating a basic simulation environment which would closely approximate real flight to adequately test our hard coded solution before actually trying it out on the physical plane.

With this simulator, we were not concerned as much with principals of lift / gravity and other such physics principles. We already know that the plane can fly well with the weight we have and we also know more or less the maneuverability of the craft given various elevon / speed levels. Thus with our simulator, we were more concerned with testing the system as a whole to make sure it reacts appropriately to all requests and fast enough that in real flight the speed of the craft and reaction time of the flight control system would be sufficient.

For further testing purposes without needing to hook up to the plane's circuit/hardware, we also created a "virtual microcontroller" object which mimics the actual hardware layer in every way line by line in code, except that it doesn't actually control any hardware. We also created in tandem a simulated communication layer which is the same as the actual communication layer in functionality but instead deals with communicating with the virtual hardware layer. Either this simulated hardware layer or the actual hardware layer can be used for testing and the functionality is the same in either case, with a slight increase in speed, though not noticeably so, with the simulated hardware due to bypassing serial communication latency and some very small latency in retrieving senor data. Neither of the latency delays in either case are significant in terms of noticeable slowdown due to the fact that typically at most two bytes of data are being sent to and from the main computer and the microcontroller and a connection is maintained for the duration of flight. Further, any code running on the microcontroller was first tested to be functional and tested to verify it would be sufficiently fast for our purposes before being translated appropriately to the virtual microcontroller object.

The final aspect of our simulator is a windows forms style environment where we could watch the state of the simulated plane / hardware. We also added a virtual 2D visual environment so we could watch the virtual plane complete the requested missions. This piece of the simulator was not strictly necessary, but was developed thinking ahead to some future add-ons we'd like to make to this project as outlined in the future work section.

6.2 Sensor Data Simulation

To accomplish this, we constructed the simulation layer such that everything is exactly as it will be when the plane is actually flying, including controlling the elevons and motors. The only difference is of course that the plane is not actually going anywhere, so the sensor data must be

realistically simulated given the motor/elevons state of the plane.

The sensory data was then simulated as follows; to simulate the pitch/yaw data from the 2D accelerometer, we based these values on state of the elevons. For adjustments in altitude, we keep track of the previous altitude and take into account the state of the elevons as the plane is flying, adjusting the altitude up or down accordingly.

The heading is adjusted in a similar fashion to the altitude based on the elevon levels throughout flight making small adjustments accordingly as the plane steps through its virtual environment.

Specifically, adjustments in pitch were simulated as follows:

$$((R + L) / 2) - 90$$

where R and L are the current degree values of the right and left elevons. This then guarantees that the pitch value returned from this equation will be between -90 and 90 and be approximately what it would be if the plane was actually flying.

For simulating the roll of the plane we start by evaluating the elevon states to determine if the plane is turning left or right or flying straight. If the plane is turning left:

$$((R - L) / 2) * -1$$

If turning right, the same equation is used except without multiplying by the negative one value. If the plane is determined to be flying straight from the elevon levels, then we simply pass back the value of zero. In this case, as with the pitch value, the returned value will always be in the range of -90 to 90 degrees and will more or less mimic the real pitch/roll values when the plane is actually flying.

Exact values matching the eventual real hardware data are not necessary for verifying the system works as it should; rather approximate values are only needed to "drive" the system in terms of making altitude/bearing adjustments and for use in the simulator front end. This is because the actual flight control system does not base its behavior on specific elevon levels corresponding with specific pitch/roll values, nor specific pitch/roll values corresponding to specific altitude or bearing adjustments. Rather the system dynamically adjusts the elevons based on the accelerometer's pitch/roll data readings and detects altitude/bearing from the GPS sensor and adjusts things accordingly from that data.

For determining the current latitude and longitude this was slightly more complicated computationally. First, we start out with a set starting latitude and longitude. From there, we base the changing of the latitude and longitude on the heading and step a set distance value (1 meter) every set time unit.

Specifically calculating the lat/lon changes as the simulator steps through execution and making additional adjustments to the heading is done as follows [13]**:**

$$lat2 = \arcsin(\cos(hdng1)*\cos(lat1)*\sin(alpha) + \sin(lat1)*\cos(alpha))$$

$$long2 = \arccos((\cos(alpha) - \sin(lat1)*\sin(lat2)) / (\cos(lat1)*\cos(lat2))) + long1$$

$$hdng2 = \arccos((\sin(lat1) - (\sin(lat2)*\cos(alpha)) / (\cos(lat2)*\sin(alpha)))$$

Here all the lat/lon values are first converted to radians from degrees; lat2 and long2 are the end adjusted coordinates; alpha is the radial distance covered on the earth's surface, which is distance traveled divided by the radius of the earth; lat1 and long1 are the original coordinates; hdgn1 and hdng2 are the original heading at the lat1 and lon1 and the final heading at lat2/long2 respectively. The final results are then converted back into degrees and after each step are then set as the new current lat/lon position of the simulated aircraft. The derivation of this formula is beyond the scope of this paper; for full details on this derivation see *Walking Around the World* from the math forum at Drexel [10].

6.3 Testing and Results

In order to actually test the flight control software, we chose to test it in the following manner. First, we tested each of the main elements of the flight control system, by generating requests to induce certain behavior in flight. These were generated to test: adjusting speed, pitch (altitude), and roll (direction) using the following Request Objects (note: default speed and altitude are automatically set to be 5 and 100 feet respectively; default lat/lon are read upon startup or in simulation set arbitrarily):

Request	Speed	Altitude	Lat	Lon	Successful?
1	10	100'	Null	Null	Yes
2	5	200'	Null	Null	Yes
3	5	100'	A	B	Yes

Figure 3: Request 1 adjusts speed to 100% from the default 50%. Request 2 adjusts the altitude to 200 feet from the default 100 feet. Request 3 points the plane towards the coordinate A/B such that A = 47.6040577439516; B = -122.328708836937 with the default coordinates set at Lat = 47.6046367558241; Lon = -122.328610395587 approximately 214 feet away and approximately 18 degrees off from the default bearing.

Having now tested each of the main elements of the Request object running through the flight control system and having had it successfully perform each action as expected, we now used the simulator itself with a set of

simple test missions. The missions are generated such that a random latitude and longitude coordinate a random small distance from a given start point is created along some random bearing. The mission is then complete when the virtual plane successfully flies itself to that point.

A successful mission is defined as one in which the simulated AUAV maintains its correct speed and altitude for the majority of the flight time and makes adjustments there as necessary and achieves each of the lat/lon request points successfully and efficiently (i.e. not going off course needlessly, but rather flying more or less directly to whatever the current destination point is).

Failure is defined as any mission where the simulated AUAV does not achieve all of the request's coordinate, speed, and altitude objectives or makes too many invalid adjustments resulting in excessive flight time due to not heading more or less directly towards the target point.

Alt.	Speed	Bearing	Distance	Lat/Lon	Successful
107	5	159.21	92.73	A	Yes
137	5	152.21	70.63	B	Yes
84	6	164.28	114.98	C	Yes
112	7	295.09	124.99	D	Yes
88	5	86.19	141.00	E	Yes
99	7	330.59	125.99	F	No
101	7	216.00	126.98	G	Yes
118	5	345.80	134.57	H	No
142	9	319.94	126.32	I	Yes

Figure 3: Altitude in feet. Speed ranges from 0-10 representing 0-100% motor power settings. Bearing is in degrees and Distance is in feet; both are the initial bearing/distance between the start/end coordinates.

	Lat. 1	Lon. 1	Lat. 2	Lon. 2
A	47.603828	122.328567	47.6035903378202	122.328960795441
B	47.603828	122.328567	47.6036566992679	122.328700866416
C	47.603828	122.328567	47.603524557482	122.328693654515
D	47.603828	122.328567	47.6039733404244	122.328106802424
E	47.603828	122.328567	47.6038536261558	122.329138965168
F	47.603828	122.328567	47.6041288789349	122.328315475262
G	47.603828	122.328567	47.6035463640001	122.328263517016
H	47.603828	122.328567	47.6041867511266	122.328432396501
I	47.603828	122.328567	47.6040944583655	122.328234732939

Figure 4: Lat1 and Lon1 represent the start coordinate. Lat2 and Lon2 represent the end coordinate. Column 1 corresponds to the rows of Figure 3 as marked by each letter.

As you can see from Figure 3 and Figure 4 all but two of the mission runs were successful. In each case the correct initial bearing was calculated as well as correctly calculating updated bearings as the plane flies through the simulator. The two test runs that were not successful were determined to have failed due to slight imprecision in the formula for calculating the latitude and longitudinal movement as the plane steps through the simulator on a given heading. This was specifically due to the extremely close, distance wise, lat/lon coordinate pairs. In this calculation, roughly one meter is stepped at a time resulting in a new "current" lat/lon based on the current heading. By stepping in such short intervals it caused some imprecision in the results of these steps which in turn resulted in the simulated plane determining an incorrect bearing and thus taking excessive time and making invalid adjustments while trying to fly to the given destination point.

This problem however, is not with the flight control system itself, but rather with the simulator. Thus, this is not an issue if the plane were actually flying itself as it would be reading that latitude and longitude information from the GPS unit instead of having a simulator generate that data given the system state.

7. Conclusion

In this research project, we developed a robust flight control system that is capable of being used on a real world Autonomous Unmanned Aerial Vehicle (AUAV). In addition to this, we covered hardware and plane requirements including setting and meeting our goal of keeping the chosen hardware/plane relatively inexpensive in order that others could more easily build off this project without needing significant funding.

The flight control system itself was built using a layered, modular approach to allow for maximal extensibility as well as allow for the system to be implemented on a variety of UAV types. We tested this flight control system using a custom built flight simulator and received favorable results; the only problems ending up being slight inaccuracies in the results of the equations that the simulator used to keep track of the current latitude and longitude of the plane as it stepped through the simulation. This was not an issue however with the flight control system, thus will not effect the plane itself during non-simulated flight.

In addition, we tested this flight control system using real hardware (servo's and motor) to verify functionality on the actual plane designed for this project. Only needing then to simulate the sensor data, as the plane was not flying during these tests.

For more information on designing and building the UAV presented in this project and to find out where you can access the code and other information on this project not included here, see section 9 of this paper.

8. Future Work

Much of the work done on this project was done in very specific ways planning ahead for some of the future work to be done. This includes shortly testing this system on the actual plane built in this project as well as testing on a couple other plane designs to determine how well it functions on a variety of platforms. The needed steps to make this happen, using the current plane and hardware, is to convert the bread boarded circuit to a more stable permanent circuit and then finally to get the appropriate FAA permits for flying an autonomous UAV.

Once working well in real flight, we would also like to add a transmitter/receiver pair on each side so that the plane can be controlled directly from a "base computer" and, with the addition of cameras streaming visual data back to the base computer, will allow for out of site flying as well as a variety of other features like integrating Google Earth / Maps and having the ability to monitor or control several planes from one work station as they fly themselves around out of range. With these cameras, we could also implement some machine vision algorithms for the planes themselves to recognize objects on the ground and notify the base station when it has found the requested object. These types of features lend themselves well to search and rescue type missions and the like.

Finally, while flying around at given altitudes, it can be generally assumed the plane will not collide with anything; however, an additional feature to add in would be an obstacle avoidance layer using two light weight cameras. This would also lend itself nicely to be able to experiment with formation flying, landing, taking off, and various swarming scenarios.

9. Acknowledgements and Notes

Special thanks to Chris Klick at RitewingRC and everyone at RCGroups.com for all their help in picking the best plane design and parts for a project of this type, as well as providing a wealth of information and guidance on building and flying the plane itself.

Additional information on the plane design and other aspects of this project including the source code for the flight control system, simulator, and microcontroller code can be found here: http://www.hiskey.us/AUAV/

10. References

[1] Gregory J. Barlow and Choong K. Oh. Robustness analysis of genetic programming controllers for unmanned aerial vehicles. In Proceedings of the 8th annual conference on Genetic and evolutionary computation, ACM Press, 135-142 2006.

[2] Fang Wang, Eric Mckenzie. A multi-agent based evolutionary artificial neural network for general navigation in unknown environments. Proceedings of the third annual conference on Autonomous Agents, ACM Press, 154-159, 1999.

[3] Frank Hoffmann, Tak John Koo, Omid Shakernia. Evolutionary Design of a Helicopter Autopilot. 3rd On-lin World Conference on Soft Computing, 1998

[4] Prithviraj Dasgupta. Distributed automatic target recognition using multi-agent UAV swarms. Proceedings of the fifth international joint conference on Autonomous agents and multi-agent systems. ACM Press, 479-481, 2006

[5] D.H. Shim, H.J. Kim, S. Sastry. A Flight Control System for Aerial Robots: Algorithms and Experiments. IFAC Control Engineering Practice. 2003.

[6] D. Shim, H. Chung, H.J. Kim, S. Sastry. Autonomous Exploration in Unknown Urban Environments for Unmanned Aerial Vehicles. AIAA GN&C Conference, 2005

[7] O. Shakernia, C.S. Sharp, R. Vidal, D.H. Shim, Y.Ma, S. Sastry. Multiple View Motion Estimation and Control for Landing on Unmanned Aerial Vehicle. Robotics and Automation, IEEE International Conference Volume 3, 2793-2798, 2002

[8] D.H. Shim, H.J. Kim, H. Chung, S. Sastry. Multi-functional Autopilot Design and Experiments for Rotorcraft-based Unmanned Aerial Vehicles. 20th Digital Avionics System Conference, Vol 1, IEEE, 14-18, 2001

[9] C. Tomlin, I. Mitchell, A. Bayen, and M. Oishi, "Computational techniques for the verification of hybrid systems"

[9] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolution*

[10] The Math Forum at Drexel, "Walking Around the World". Ask Dr. Math. 8 February, 1997. <http://mathforum.org/library/drmath/view/52049.html>

[11]. *Numerous Authors*. “Stanley: The Robot that won the DARPA Grand Challenge.” Journal of Field Robotics 23(9), Wiley Periodicals, Inc. 661-692, 2006. <http://robots.stanford.edu/papers/thrun.stanley05.pdf>

[12] D.H. Shim, H.J. Kim, S. Sastry. FLYING ROBOTS: Sensing, Control, and Decision Making. IEEE International Conference on Robotics and Automation. 2002.

[13] P.O. Pettersson, P. Doherty. “Probabilistic Roadmap Based Path Planning for an Autonomous Unmanned Aerial Vehicle.”

[14] Nidal M. Jodeh, “Development of Autonomous Unmanned Aerial Vehicle Research Platform: Modeling, Simulating, and Flight Testing.” Department of the Air Force Air University. 2006

[15] K.D. Mullens, E.B. Pacis, S.B. Stancliff, A.B. Burmeister, T.A. Denewiler, M.H. Bruch, H.R. Everett. “An Automated UAV Mission System.”